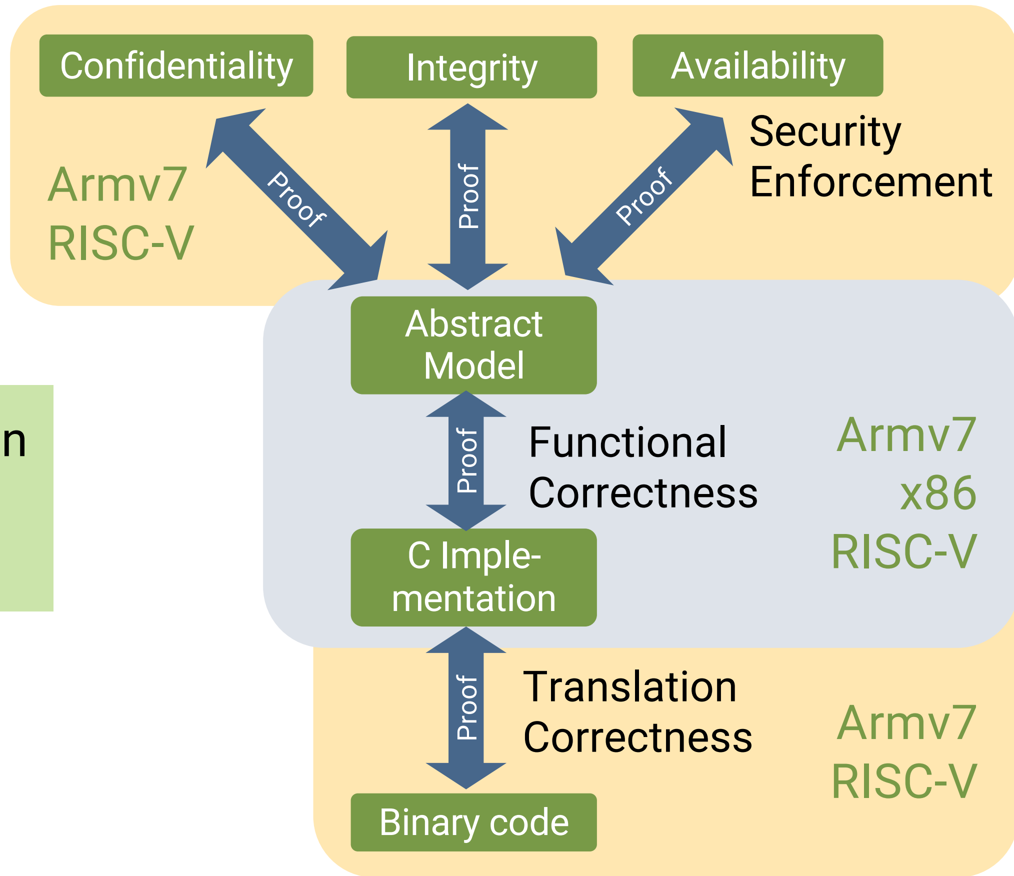
School of Computer Science & Engineering

Trustworthy Systems Group

KISS: Making Dependable Operating Systems a Reality

Gernot Heiser
gernot@unsw.edu.au

We Have seL4



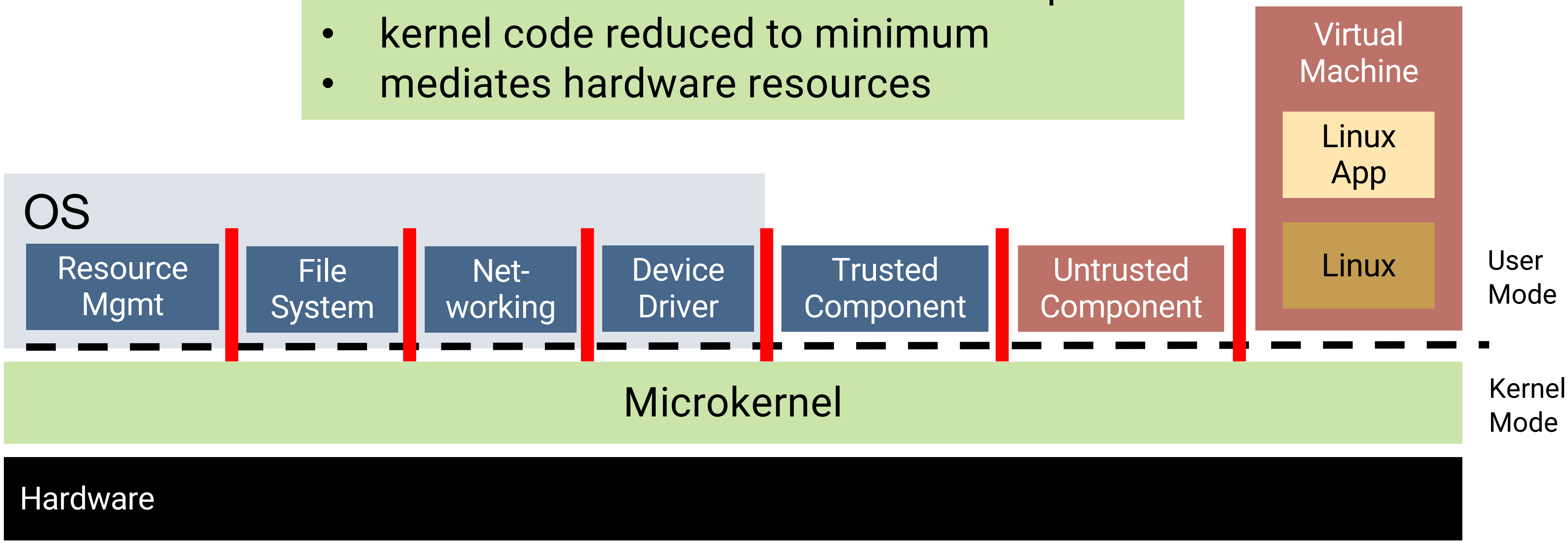
- Comprehensive formal verification
- Provable real-time capability
- World's fastest microkernel

Present limitations
- initialisation code not verified
- MMU, caches modelled abstractly
- Multicore not yet verified

Microkernel Is Not An OS

Modularisation: Separate components
- operating-system services
- applications

Microkernel enforces isolation – bullet-proof
- kernel code reduced to minimum
- mediates hardware resources



Can We Build A Verified OS?

… where the whole trusted computing base is proved correct?

I Claim We Can!

… if we strictly observe some fundamental principles: KISS

- Fine-grained modularity, strong separation of concerns
- Least privilege
- Simple abstractions
- Simple policies
- Simple implementation

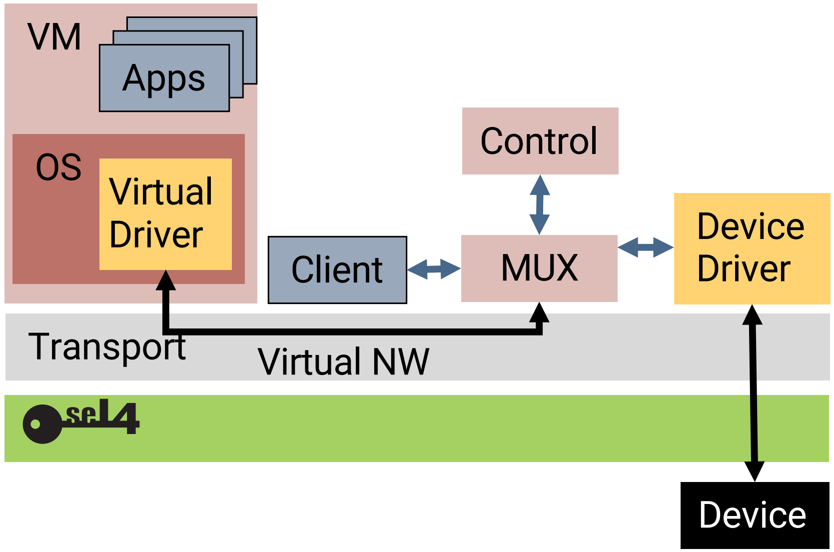
Reason about security

Enables verifying modules separately

- "Universal" policies are complex & have pathological cases
- Better use-case-specific, swappable policies
- Requires policy modularity

- Enabled by the above
- Enable push-button verification!

Key Component: Driver Framework

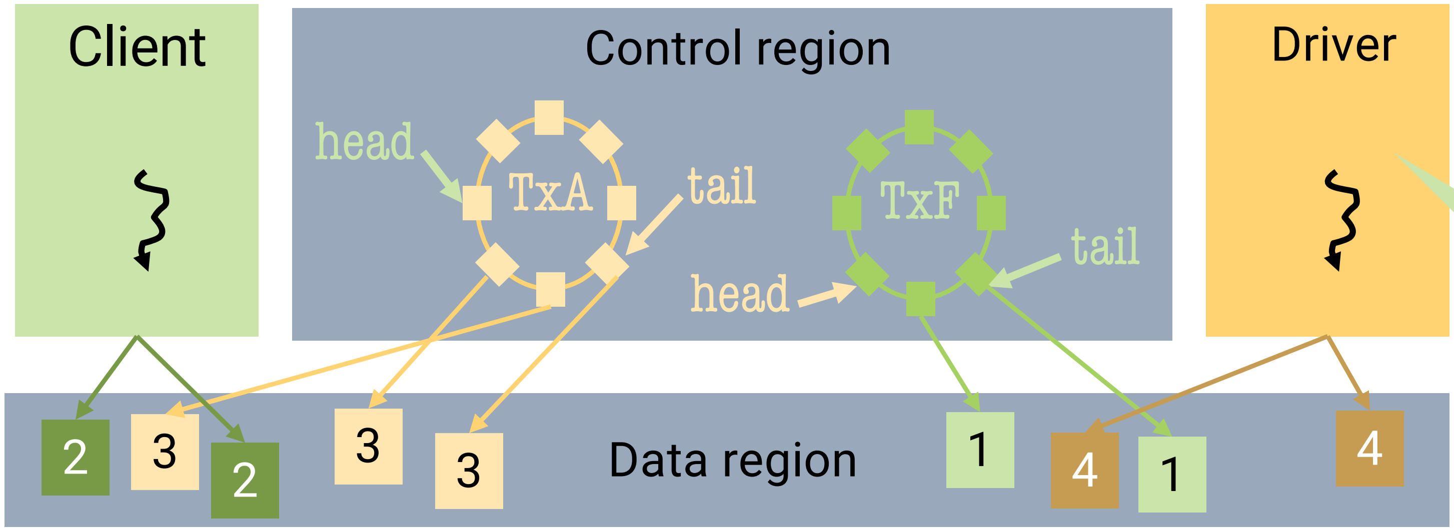


Aim:
- Simple model for robust drivers
- Secure, low-overhead sharing of devices between components
- Low overhead

Approach:
- Zero-copy transport layer
- Each component simple, single-purpose
- Standard interfaces, virtIO

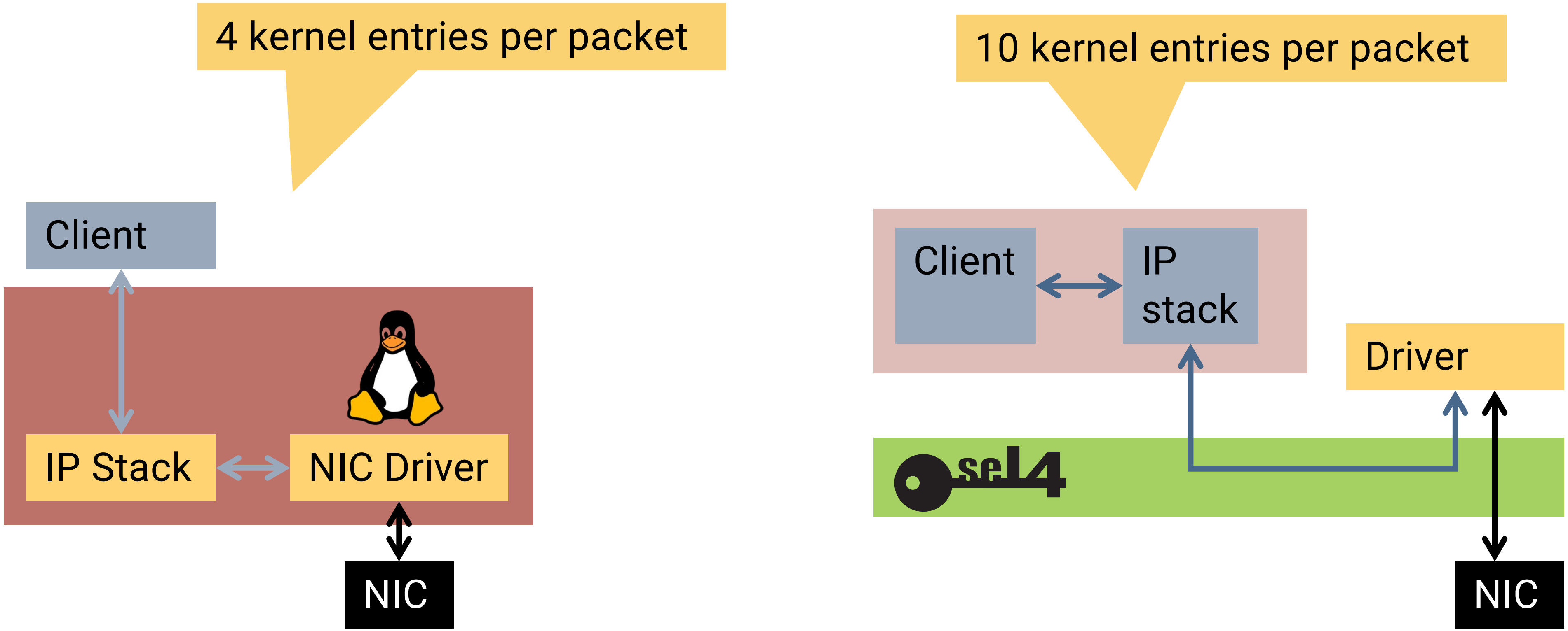
seL4 Device Driver Framework (sDDF)

- Lightweight
- Separation of concerns: driver only translates interfaces
- Simple, event-based, single-threaded drivers
- Asynchronous, zero-copy transport layer
- Bounded, lock-free, single-producer, single-consumer queues

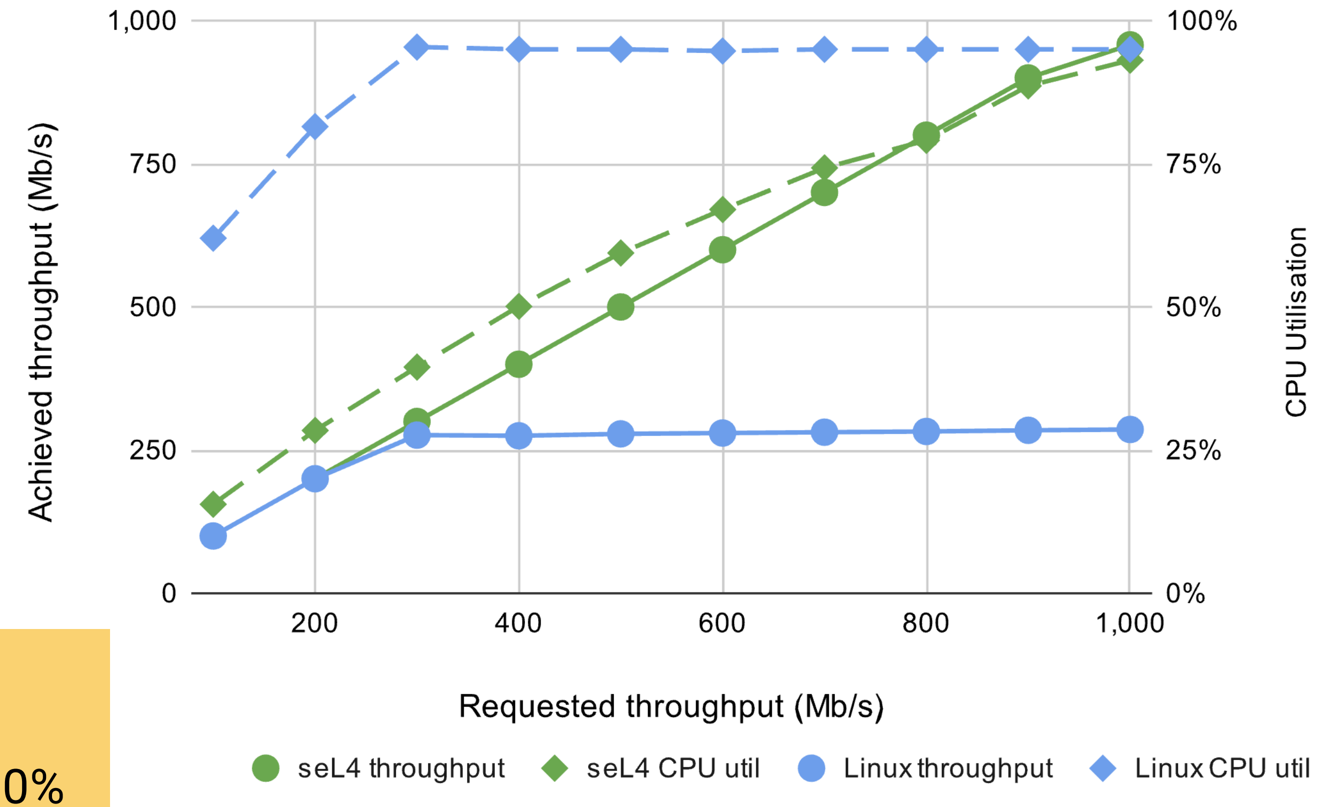


Example: Gb/s NIC
- Linux : 3,000 LoC
- seL4 : 500 LoC

Performance Evaluation Setup

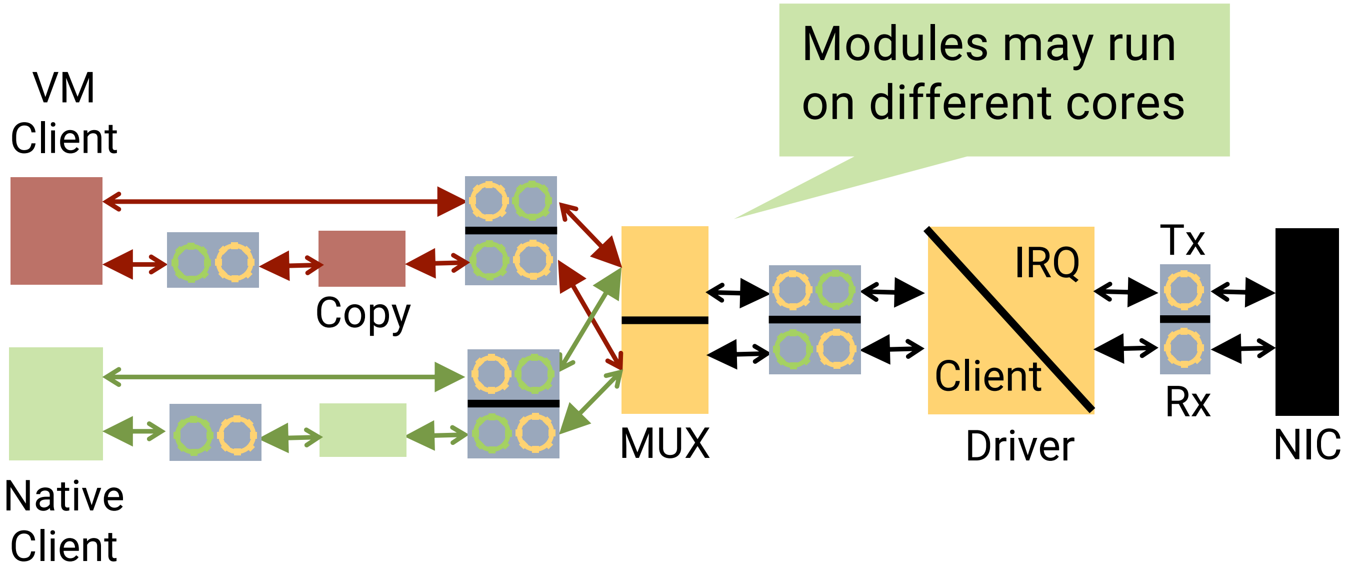


seL4 vs Linux Networking Performance



- Outperforms Linux
- Overhead of extra domain crossing ≤10%

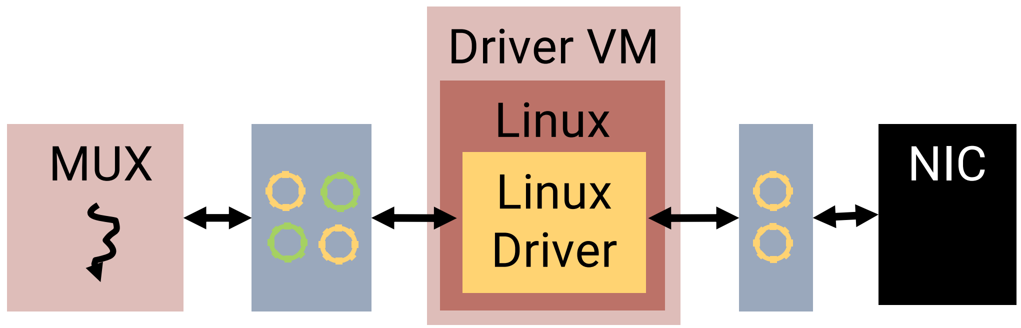
Full Network System



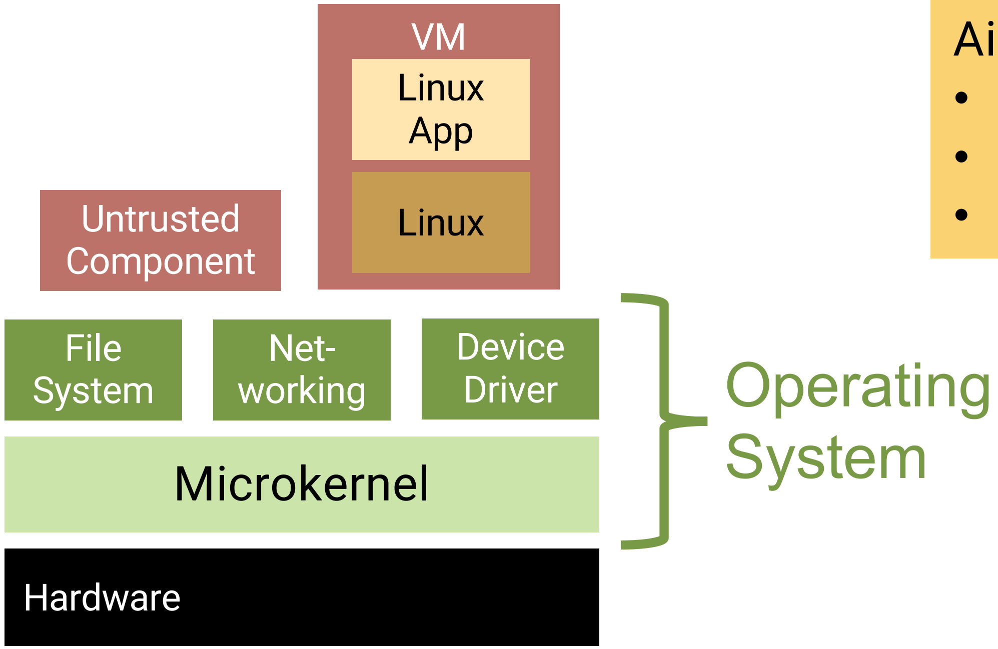
- Each component is simple & single-threaded
- Most split into separate Tx/Rx modules
- Copy where needed for security
- IP stack is client library, only handles UPD & TCP
- Broadcasts, DHCP handled by separate modules

Legacy Re-Use

- Can use Linux drivers wrapped into individual driver VM



OS = Kernel + Drivers + I/O Services



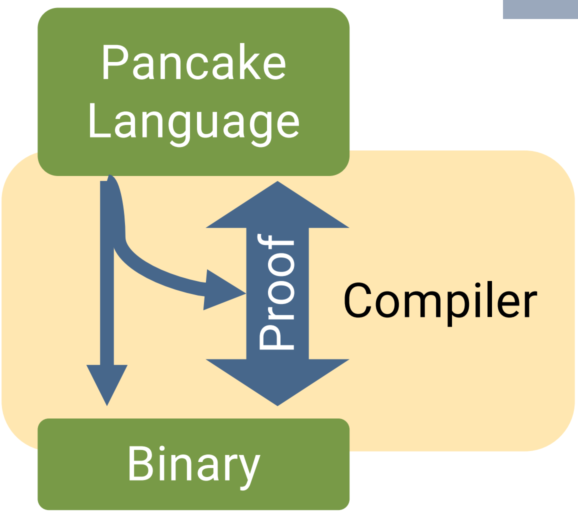
Aim: Verified OS for Cyberphysical/IoT
- Highly modular design
- Simple component implementation
- Performant

- Most components just a few 100 LoC, sequential
- Can use push-button techniques (SMT solvers)

Trustworthiness: Verification-Friendly Systems Language – Pancake

Reducing Cost of Verified Systems Code

Aim: Simplify verifying user-level OS components



Idea:
- Use low-level but safe systems language with certifying compiler
- Gives many proof obligations for free

Systems language:
- memory safe
- not managed (no garbage collector)
- low-level (obvious translation)
- interfacing to hardware
- no run-time system

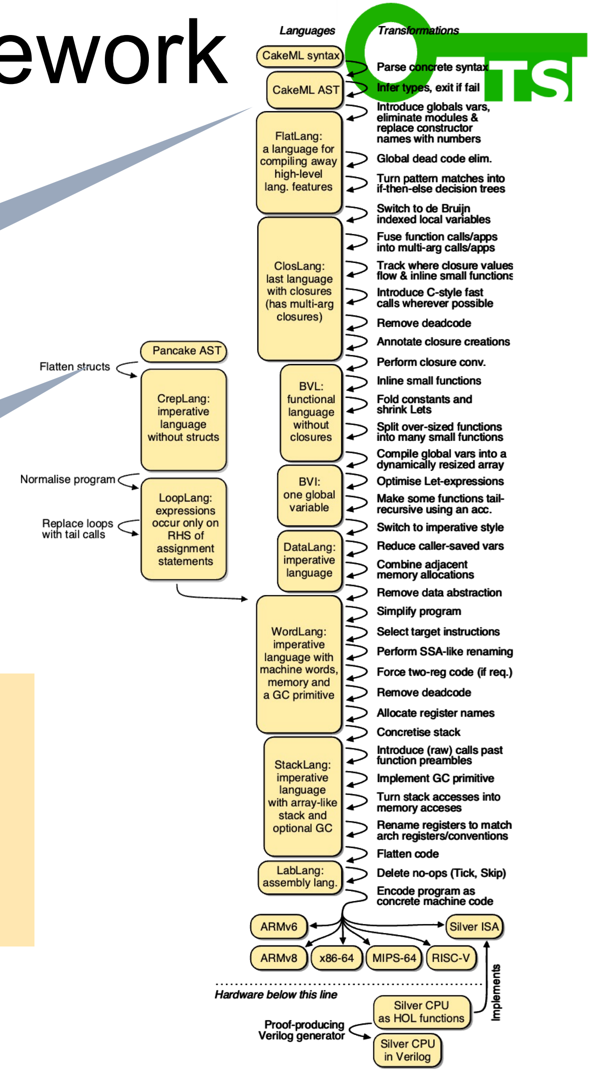
Approach: Re-Use CakeML Framework

CakeML:
- functional language
- type & memory safe
- managed (garbage collector)
- high-level, abstract machine
- verified run time
- verified compiler
- mature system
- active ecosystem

Great, but too high-level!

CakeML compiler

Pancake compiler



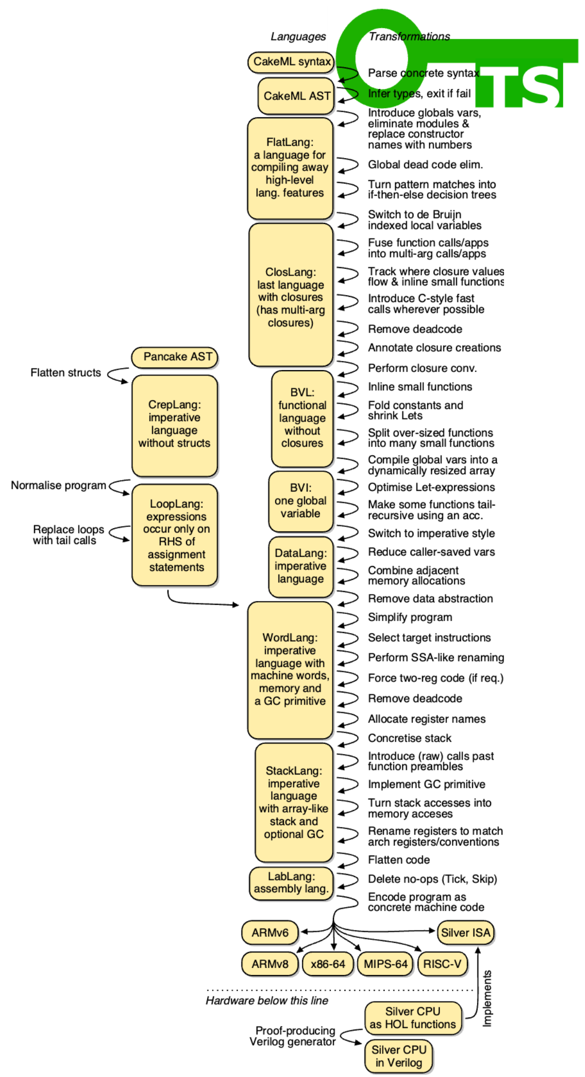
Approach:
Re-use lower part of CakeML compiler stack for imperative language

Verified Pancake Compiler

Pancake compiler is written in CakeML
⇒ can use CakeML compiler to produce verified Pancake compiler binary!

Status:

- Mostly done: Toy (serial) driver verification to explore semantics
- Prototype done: Parser
- Almost done: Verification of link to CakeML compiler:
- In progress: Binary compiler bootstrap
- Not started: Shared-memory driver-device, driver-client



UNSW SYDNEY

Summary

I'm confident we can build an seL4-based OS that:

- has sufficient functionality for real-world IoT/cyberphysical systems
- outperforms Linux
- has a verified trusted computing base

School of Computer Science & Engineering

Trustworthy Systems Group

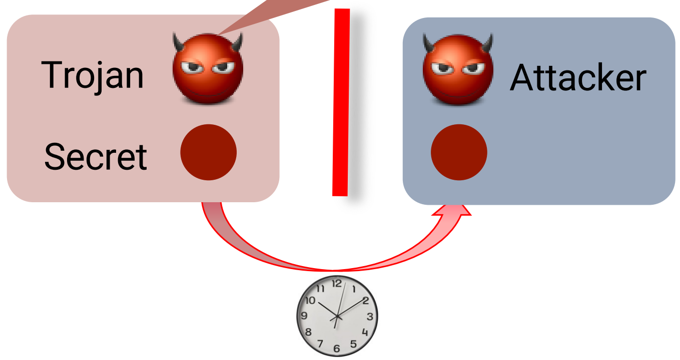
Time Protection: Principled Prevention of Microarchitectural Timing Channels

Gernot Heiser
gernot@unsw.edu.au

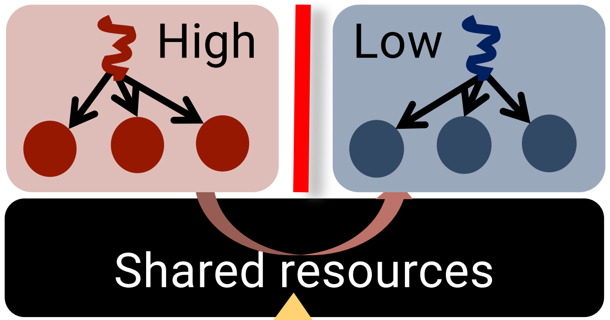
Covert Timing Channels

Spectre attack shows Trojans can even be constructed in innocent code!

May hide in any code not proved correct!



Microarchitectural Timing Channels

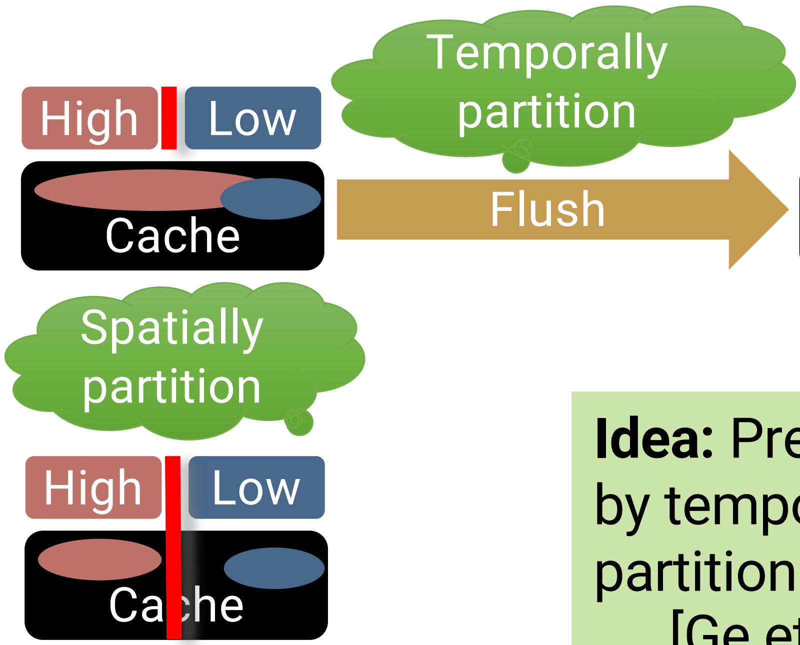


High affects Low's progress
- Information leakage
- **Confidentiality violation**

Microarchitectural timing channels: Contention for shared hardware resources affects execution speed

Standard approach: Patch & Pray

Time Protection: Principled Prevention



Aim: *Provably prevent* information flow through micro-architectural timing channels

Idea: Prevent channels by temporal or spatial partitioning of **all** HW [Ge et al, EuroSys'19]

Temporal Partitioning: Flush on Switch

Must remove any history dependence!

1. T_0 = current_time()
2. Switch user context
3. Flush on-core state
4. while (T_0+WCET < current_time()) ;
5. Reprogram timer
6. return

Latency depends on prior execution!

Time padding to remove dependency

Proving Temporal Partitioning

Must remove any history dependence!

Prove: flush all non-partitioned HW
- Needs model of stateful HW
- Somewhat idealised on present HW … but matches RISC-V prototype
- **Functional property**

1. T_0 = current_time()
2. Switch user context
3. Flush on-core state
4. while (T_0+WCET < current_time()) ;
5. Reprogram timer
6. return

Prove: access to shared data is deterministic
- Each access sees same cache state
- Needs cache model
- **Functional property**

Prove: padding is correct

Padding: Use Minimal Clock Abstraction

Abstract clock = monotonically increasing counter
Operations:
- Add constant to clock value
- Compare clock values

To prove: padding loop terminates as soon as clock ≥ T0+WCET
- **Functional property!**

Time Protection Verification: Status

1. [Done] Specify isolation property
2. [Done] Prove enforcement on high-level model
3. [In progress] Connect to seL4 proofs
    1. [Done] Update seL4 abstract specification to account for memory accesses
    2. Prove these accesses are bounded according to security policy
    3. Connect 3.1-3.2 to high-level model to prove isolation property
    4. Prove preservation of 3.1-3.3 by refinement to lower-level seL4 specifications

Hardware Support for Time Protection

1. T_0 = current_time()
2. Switch user context
3. Flush on-core state
4. while (T_0+WCET < current_time()) ;
5. Reprogram timer
6. return

Hardware Reality:
Mainstream processors do not allow resetting all history-dependent state!
[Ge et al., APSys’18]

RISC-V to the rescue!
- Add instruction to clean state
- Also help with padding

[Wistoff et al, DATE’21]

Defining the state of the art in trustworthy systems since 2009